WinUI FUNDAMENTALS:

(2) WHAT IS WinUI:

- NATIVE UI FRAMEWORK OF WINDOWS 10/11 APPS.
- DESKTOP APPLICATIONS IN C# OR C++.
- WinUi SHIPS AS PART OF THE WINDOWS APP SDK.
- WinUI FEATURES:
  1) XAML (EXTENSIBLE WINDOWS MARKUP LANGUAGE)
  2) DATA BINDING (BINDS DATA TO UI).
     - PART OF THE MVVM PATTERN.
  3) LAYOUT
  4) STYLES AND TEMPLATES

* CREATING A WinUI PROJECT:

- MAIN WINDOW:
  1) MainWindow.xaml → UI
  2) MainWindow.xaml.cs → CODE BEHIND IT

- APPLICATION OBJECT:
  1) App.xaml
  2) App.xaml.cs

* XAML AND CODE-BEHIND FILE:

- GRID (LAYOUT PANEL) - MOST USSUAL ROOT PANEL.
- BUTTON - AN BUTTON ELEMENT.
  ATRIBUTES:
  1) CONTENT → Value inside the BUTTON
  2) BACKGROUND → Color of the BUTTON BACKGROUND.
  3) VERTICAL ALIGMENT → SETS THE POSSITION OF THE BUTTON ON THE VERTICAL Axis.
  4) HORIZONTAL ALIGMENT → SETS THE POSITION OF THE BUTTON ON THE HORIZONTAL Axis.

5) MARGIN → NUMBER OF PIXEL THAT PUSH THE ELEMENT FROM OTHER ELEMENTS.

6) Click → NAME OF THE ON CLICK EVENT HANDLER.

7) x:NAME → Unique NAME FOR THE XAML ELEMENT. USED FOR ACESSING IT in THE CODE BEHIND FILE.

#UNDERSTANDING how FILES ARE GENERATED:

- PARTIAL CLASS MEANS THAT A CLASS DEFINITION CAN SPAN ON MULTIPLE FILES.
- PART OF THE PARTIAL CLASS IS GENERATED BY THE .XAML FILE. In here is the „InitializeComponent()" METHOD AND THE ELEMENT PROPERTIES THAT HAVE A UNIQUE x:NAME VALUE.
- When THE REST OF THE PARTIAL CLASS IS GENERATED IT USES THE x:Class ATRIBUTE TO GENERATE THE NAMESPACE AND CLASS NAME.

[.XAML]

x:Class="App.CsApp.MainWindow"

[MainWindow.G.I.CS] → GENERATED FILE with THE REST OF THE PARTIAL CLASS.

NAMESPACE App.CsApp
class MainWindow

3 INSTANCING OBJECTS in XAML:

#WORK with ELEMENTS AND ATTRIBUTES:

- element → BUTTON ETC.

<BUTTON> ↓ element

</BUTTON> ↓ closing element

-<BUTTON CONTENT="ADD CUSTOMER" click="e_click" />

↓ ATRIBUTE is MAPPED TO A ATRIBUTE

↓ ATRIBUTE MAPPED TO AN EVENT.

↓ Element is MAPPED TO A CLASS.

* Set Properties with Property Element Syntax:

<Button> ⟶ Maps to Button class and creates object of type "Button".

<Button.Content>
Add Customer!
<Button.Content/> ⟶ Property element, maps to Content property of the Button class.

</Button>

- Example:

```
<Button>
    <Button.Content>
        <StackPanel>
            <SymbolIcon Symbol="Add">
            <TextBlock Text="Add Customer">
        </StackPanel>
    </Button.Content>
</Button>
```

Creates a Button with complex Content.

- Margin = "5 0 0 0" ⟶ Bottom
  - 5 → Left
  - 0 → Top
  - 0 → Right

* Set properties with Content Syntax:

- example:

```
<Button>
    Add Customer!
</Button>
```

↗ Content inside the element tags.

* XAML PARSER (READS THE XAML DOC):

- CREATES OBJECTS FROM THE XAML DOCUMENT.
- HOW IT WORKS in the EXAMPLE ABOVE:
  1) CONTENT without PROPERTY element
  2) looks FOR the CONTENTPROPERTYATTRIBUTE
  3) [ContentProperty(Name = "Content")]
     Public class ContentControl → BUTTON BASE class
  4) Uses the PROPERTY SPECIFIED in the ATTRIBUTE.

* Collection syntax:

```
<StackPanel>
    <StackPanel.Children>
        <TextBlock/>
        <Button/>
    </StackPanel.Children>
</StackPanel>
```

→ Collection syntax

XAML PARSER:
```
VAR StackPanel = new StackPanel();
StackPanel.Children.Add(New TextBlock());
StackPanel.Children.Add(New Button());
```

4. Building a User Interface:

*Know the WinUI layout Panels:

- To arrange and position elements in your UI, we use layout panels.
- A layout panel is a container for your elements.
- The layout panels are: StackPanel, Grid, Canvas.

1) StackPanel:

```
<StackPanel Orientation="Horizontal">

    <Rectangle Fill="LightBlue" Height="20" Margin="2"/>
    <Rectangle Fill="LightBlue" Height="20" Margin="2"/>

</StackPanel>
```

- Stacks children vertically by default, for horizontal use Orientation atribute.
- height width → value in pixels.
- Great for organizing simple things.

2) Grid

- The most powerful layout panel.
- Used to arrange elements in the main window.

```
<GRID
    <GRID.RowDefinitions>  ──→ DEFINES ROWS
        <RowDefinition/>
        <RowDefinition/>
    </GRID.RowDefinitions>
    <GRID.ColumnDefinitions> ──→ DEFINES COLUMNS
        <ColumnDefinition width="40" />
        <ColumnDefinition/>
    </GRID.ColumnDefinitions>
    <Rectangle Fill="LightBlue" Grid.Column="1" Grid.Row="1"
</GRID>
```

SECOND COLUMN AND ROW

- GRID.COLUMN → ATTACHED PROPERTY, GRID PROPERTY THAT IS NOT SET ON THE GRID ITSELF. IT IS ZERO INDEXED.

3) CANVAS:

- USED FOR ABSOLUTE POSITION OF ELEMENTS.

```
<Canvas>
    <Rectangle Fill="LightBlue" Height="50" Width="50"
    Canvas.Left="50"
    Canvas.Top="100"/>
    <Rectangle Fill="Orange" Height="50" Width="50"
    Canvas.Left="75"
    Canvas.Top="125"/>
</Canvas>
```

→ THE ORANGE RECTANGLE WILL BE ON TOP OF THE BLUE ONE.

- CANVAS.LEFT → MOVES ELEMENT 50 PIXELS FROM THE LEFT.
- CANVAS.TOP → MOVES ELEMENT 100 PIXELS FROM THE TOP.

- The XML DOCUMENT IS RED TOP TO BOTTOM, so the ORANGE RECTANGLE is on TOP OF the BLUE one. To CHANGE THAT we CAN SET CANVAS.ZINDEX="1" (DEFAULT FOR FIRST element is 0).

*BUILD A LAYOUT with the GRID:

- When ADDING IMAGES SET BUILD ACTION TO CONTENT.
- XAML live PREVIEW DEBUG → WINDOWS
- GRID.COLUMNSPAN / GRID.ROWSPAN → SETS how MANY „BLOCK" will the element TAKE.

*UNDERSTAND the SIZE OF ROWS AND COLUMNS:

1) <RowDefinition Height="*" />
   <RowDefinition Height="*" /> → SAME SIZE ROWS

2) <RowDefinition Height="3*" /> → 75% OF the SPACE
   <Row Definition Height="*" /> → 25% OF the SPACE

- This is STAR SIZING.
- Absolute sizing is ANOTHER METHOD

<Row Definition Height="100" /> → WONT GROW OR SHRINK WHEN RESIZING
<Row Definition Height="*" />
<Row Definition Height="*" /> → SHARE the REST OF the SPACE equally.

↓

CAN BE SET TO "AUTO", then the height is OF the highest element OF THAT ROW.

<RowDefinition Height="100" />
<RowDefinition Height="*" />
<RowDefinition Height="Auto" />

↓

In ATRIBUTE SYNTAX:

↓

<GRID RowDefinitions="100,*,Auto" />

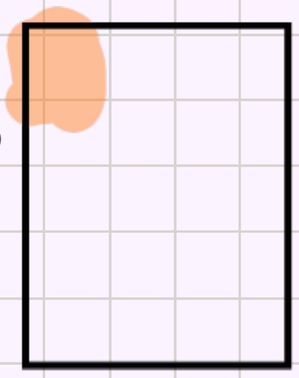
→ In this exampe the top and bottom rows will be fixed and the middle row will resize based on the window.

↓

HEADER / MAIN / FOOTER

*Use Layout Properties to Position Elements:

```
<GRID wioth="200" height="200">
    <BUTTON Content="Ok"
            Horizontal Aligment="Left"
            Vertical Aligment="Top" />
</GRID>
```



- Horizontal Aligment sets the position on the horizontal axis. Can also be set to STRETCH, sets the BUTTON to the AVAIVABLE width.
- Vertical Aligment sets the position on the vertical axis. Can also be set to STRETCH to set it to the AVAIVABLE height.
- If there is a height or width ATTRIBUTE it will always be that height and width.

- If the StackPanel has a set Orientation, setting the Aligment for the elements inside won't do Anything.

*Centering the header:

*Create a nested Grid for the Navigation:

*Set Attached Properties in XAML:

-Atribute syntax: <Button Grid.Row="1" />

-Property element syntax: <Button>
<Grid.Row>
1
</Grid.Row>
</Button>

*Set attachment properties in C#:

```
VAR BTN = new Button();
BTN.SetValue(Grid.RowProperty, 1);
```

this method is available at every element in WinUI.

RowProperty → key

1 → value

```
VAR ROW =(int)BTN.GetValue(Grid.RowProperty);
```

RETURNS the value of the row property.

* Access Attached Properties with static methods:

- VAR column = Grid.GetColumn(customerListGrid);
  - GetColumn → returns int
  - getting value

- Grid.SetColumn(customerListGrid, newColumn);
  - sets the value

5. Organizing code with UserControls:

* Extract the Header into a UserControl:
* Understanding the XAML namespaces of WinUI:

6. Applying DATA BINDING AND MVVM:

* Bind to another Element:

<TextBox Header="FirstName" Text="{Binding ElementName=CustomerView, Path=SelectedItem.Content, Mode=TwoWay}

- Binds to another value
- x:Name of ListView element
- Path

TwoWay → source changes destination and reverse
oneWay → source changes destination
oneTime → values from source are taken once.

* Know how the Data Context Works:

- <TextBlock
  Text="{Binding Source=...}" />
  - ↓ OR ↓
  - Binding RelativeSource=
    {RelativeSource Self}, Path=Width
    - Points to itself
    - Binds the value to its width.

- DataContext

```
<Grid DataContext="Pluralsight">
    <StackPanel DataContext="Thomas">
        <TextBlock Text="{Binding}" />
    </StackPanel>
</Grid>
```

→ Value is Thomas

* The Model View ViewModel (MVVM) Pattern:

- Stands for:

Model	View	ViewModel

- State of the art pattern for WinUI Apps.
- Seperates the UI from the UI logic.
- Increases testability.

1) View:

- Consist of .xaml and .xaml.cs
  MainWindow.xaml
  MainWindow.xaml.cs

- Contains UI elements like listView etc, like a customer list.

2) Model (Customer class)

- Defines the data our application needs.
- Has values like FirstName and LastName.
- Usually has a DataProvider (CustomerDataProvider) used to load and save customers. (Api, DB and etc.)

3) ViewModel class (MainViewModel):

- contains the user interface logic.
- Defines the Model values as the view needs them.
- ViewModel is using Model to get data.
- ViewModel is bound to View with a DataContext.

* Create a MainViewModel:

- CustomerDataProvider is like a service layer method for getting data for the ViewModel.
- MainViewModel:
  - ObservableCollection<object> - special collection that notifies the data binding when items are added or removed.
  - The viewModel gets the data for the View.

*Using the ViewModel for the MainWindow:

* Add a selectedCustomer property:

*Notify About Property changes:

*Refactor Logic into a ViewModelBase class.

* Create CustomerItemViewModel:

*Use x:Bind Instead of Binding:

- Binding → resolves at compile time.
- x:Bind → resolves at runtime.
- Advantages of x:Bind:
  1) Better performance

2) Compile-time errors
3) Better DEBUG.

- x:Bind uses the root object of the XAML doc.

* Bind the Visibility with x:Bind:

7) Execute Code with Commands: